VI.15.

Тестирование и рефакторинг.

Модульное тестирование (Unit testing). Инструменты Go для модульного тестирования: package testing, testing.T.

Табличное (модульное) тестирование

Тестирование и рефакторинг.

Что есть тестирование?

Странный вопрос, вроде всё понятно, но всё-таки на всякий случай начнём с самого начала, ab ovo

> **Тестирование - это процесс исполнения программы с целью обнаружения в ней ошибок.**

В замечательной книге Керниган, Пайк. “Практика программирования” формулируется ещё образнее:

> **Тестирование - это систематические и настойчивые попытки сломить сопротивление норм ально работающей программы и заставить её сделать ошибку (Керниган, Пайк)**

Если что, то книга лежит в материалах в *7*-му занятию *III* семестра

И опять-таки для входа в занятие и для некоторого повторения, сформулируем некоторые принципы тестирования программ.

Принципы тестирования.

Принципы тестирования — это основы тестирования. Их нельзя изменить, отменить, понимать или принимать частично. В то же время однозначных формулировок принципов тестирования как неких законов природы не существует, по крайней мере пока. И формулировки принципов, и их количество подвержено изменениям со временем, но сформулируем что-то, просто чтобы тупо зацепитьься за тему. Впрочем, наформулировано тут довольно много.

- тестирование демонстрирует наличие дефектов
    - программ без дефектов не существует; программма без дефектов - это такое теоретическое абстрактное понятие
    - тестирование может показать, что дефекты присутствуют; тестирование не может доказать, что дефектов нет
    - одна из задач тестирования - проверить адекватность структур исходных и промежуточных данных структуре алгоритма
- исчерпывающее тестирование - тестирование всех комбинаций ввода и предусловий - физически невозможно
- начинать тестирование надо как можно раньше - с самого начала жизненного цикла программы
    - начинать тестирование нужно для того, чтобы найти ошибки.
    - тестировании следует вести в течение всего времени жизни программы
    - раннее тестирование сохраняет время и деньги
- большинство ошибок находятся в небольшом количестве модулей системы
    - чем больше ошибок обнаружено в каком-то фрагменте программы, тем выше вероятность того, что там есть ещё необнаруженные ошибки.
- тест обязан включать в себя ожидаемую реакцию, в частности, ожидаемые значения выходных данных
    - необходимо проверять не только, выполняет ли программа свое предназначение, но и не делает ли она того, чего должна не делать
- после исправления ошибок, выявленных на каком-то тесте, следует повторять тестирование на тех тестах, которые уже прошли успешно - исправления могут внести новые ошибки.

В общем, **тестирование - это тот стержень вокруг которого крутится не только разработка программы, но и весь её жизненный цикл, начиная с постановки задачи, и вплоть до развития и сопровождения**

Рефакторинг

Рефакторинг — это переработка исходного кода программы, чтобы он стал более простым и понятным.

Рефакторинг не меняет поведение программы, не исправляет ошибки, не повышает эффективность кода и не добавляет новую функциональность. Он делает код более понятным и удобочитаемым.

Рефакторинг – не оптимизация, хотя и может быть с нею связан. Часто его проводят одновременно с оптимизацией, но у этих процессов разные цели: цель оптимизации – улучшение производительности программы, цель рефакторинга – улучшение понятности кода. После оптимизации исходный код может стать сложнее для понимания.

После рефакторинга программа может начать работать быстрее, но это побочный эффект, главное – её код становится проще и понятнее. Стройный, хорошо структурированный код легко читается и быстро дорабатывается. Но редко удаётся сразу сделать его таким. Разработчики спешат, в процессе могут меняться требования к задаче, тестировщики находят баги, которые нужно быстро исправить, или возникают срочные доработки, и их приходится делать второпях.

В результате даже изначально хорошо структурированный исходник становится беспорядочным и непонятным. Программисты знают, как легко завязнуть в этом хаосе. Причём неважно, чужой это код или собственный. Чтобы решить все эти проблемы, делается рефакторинг программы.

В новом проекте рефакторинг нужен, чтобы:

- сохранить архитектуру проекта, не допустить потери структурированности;
- упростить будущую жизнь разработчиков, сделать код понятным и прозрачным для всех членов команды;
- ускорить отладку и разработку.

Но любое приложение со временем устаревает: язык программирования совершенствуется, появляются новые функции, библиотеки, операторы, делающие код проще и понятнее. То, что год назад требовало пятидесяти строк, сегодня может решаться всего одной. Поэтому даже идеальная когда-то программа со временем требует нового рефакторинга, обновляющего устаревшие участки кода.

Программный код предназначен не только для компьютера, но и для разработчика - человека, который будет его развивать и сопровождать.

Рефакторинг, как и тестирование, - есть необходимый элемент разработки. Опасно делать рефакторинг не постоянно, а от случая к случаю. Рефакторить надо постоянно, ни в коем случае нельзя перетряхивать сразу весь код или его существенную часть.

Суть рефакторинга и его цели посмотрели. Можно говорить о признаках того, что пора делать рефакторинг, о методах рефакторинга, о его узких местах, но это уведёт нас достаточно далеко, а у нас другие цели, так что на этом остановимся.

Тестирование и рефакторинг

Что же у них общего, как связаны тестирование и рефакторинг?
Во-первых, оба эти процесса носят продолжительный характер, да, тестирование начинается сразу, с момента постановки задачи, а рефакторинг начинается только с момента начала кодирования, но оба они не заканчиваются, пока прогрfмма вообще живёт и развивается, до самого конца её жизненного цикла.
И второе: в процессе рефакторинга самое главное - ничего не нарушать, а это обеспечивается как раз системой хороших тестов, т.е. качественный рефакторинг без качественного тестирования просто невообразим.

Увы, хоть как-то автоматизировать рефакторинг - это дело будущего, понятно, что полностью его автоматизировать просто невозможно, так что о рефакторинге - это всё, главное - мы знаем, что рефакторить надо, причём постоянно, но понемногу.
А вот с автоматизированием тестирования, да и вообще с теорией и практикой тестирования всё обстоит гораздо более продвинуто. Одна терминология, связанная с тестированием, чего стОит... Но не будем туда лезтть. Мы подробно поговорим об, видимо, основном, по крайней мере для нас, виде тестирования - о модульном тестировании (unit testing): о том, что это такое, зачем оно нужно и как его делать, о средствах, которые Go предоставляет для (автоматизации) модульного тестирования.

Модульное тестирование (Unit testing).

Что такое модульное тестирование?

Модульное тестирование (Unit testing) – это тип тестирования программного обеспечения, при котором тестируются отдельные единицы (модули, компоненты) программного обеспечения. Его цель заключается в том, чтобы обнаружить факты некорректного выполнения отдельных единиц программного кода. Это тестирование исполняет разработчик на этапе кодирования приложения. Модульные тесты изолируют часть кода и проверяют его работоспособность. Единицей тестирования может быть отдельная функция, метод, тип, пакет и т.д.

Отсутствие модульного тестирования (упаси, господи) при написании кода значительно увеличивает уровень дефектов при дальнейшем (интеграционном или системном) тестировании. Ну, из общих соображений, чем раньше выявляются какие-то неприятности, тем быстрее и легче, и, соответственно, дешевле, можно их исправить.

Зачем нужно модульное тестирование?

1. Модульные тесты позволяют исправить ошибки на ранних этапах разработки и снизить затраты.
2. Модульное тестирование помогает разработчикам лучше понимать кодовую базу проекта и позволяет им быстрее и проще вносить изменения в продукт.
3. Хорошие юнит-тесты служат проектной документацией.
4. Модульные тесты важны для миграции кода. Просто переносим код вместе с тестами в новый проект и корректируем код, пока тесты не запустятся снова.

Достоинства модульного тестирования

- Разработчики, желающие узнать, какие функциональные возможности предоставляет модуль и как его использовать, могут взглянуть на модульные тесты, чтобы получить общее представление об API модуля.
- Модульное тестирование позволяет программисту выполнить рефакторинг кода на этапе тестирования и убедиться, что модуль все ещё работает правильно. Процедура заключается в написании контрольных примеров для всех функций и методов, чтобы в случае, если изменение вызвало ошибку, его можно было быстро идентифицировать и исправить.
- Мы можем тестировать части проекта, не дожидаясь завершения других. Мы можем создавать фиктивные объекты для тестирования фрагментов кода, которые еще не являются частью законченного приложения. Фиктивные подставные объекты заменяют недостающие части программы, выполняя их роль.
- Разработчик может изолировать единицу кода для более качественного тестирования. Эта практика подразумевает копирование кода в собственную среду тестирования. Изоляция кода помогает выявить ненужные зависимости между тестируемым кодом и другими модулями или пространствами данных в продукте.

Недостатки модульного тестирования

- Не выявит всех ошибок. Невозможно оценить все пути выполнения даже в самых тривиальных программах.
- Модульное тестирование по своей природе ориентировано на единицу кода. Следовательно, оно не может отловить ошибки интеграции или ошибки системного уровня.

Рекомендации по модульному тестированию

- Модульные тесты должны быть независимыми. В случае каких-либо улучшений или изменений в требованиях, тестовые случаи не должны меняться.
- Тестируйте только одну единицу кода (модуль) за раз.

- Следуйте четким и последовательным соглашениям об именах для ваших модульных тестов.
- В случае изменения кода в каком-либо модуле убедитесь, что для модуля имеется соответствующий тестовый пример, и модуль успешно проходит тестирование перед изменением реализации.
- Исправьте все выявленные ошибки перед переходом к следующему этапу.
- Тестирование происходит одновременно с кодированием. Фактически это означает, что тесты - это часть кода. Чем больше кода вы пишете без тестирования, тем больше сценариев вам придется проверять на наличие ошибок в дальнейшем.

Инструменты Go для создания и выполнения модульных тестов на основе кода.

Язык Go имеет свой собственный набор инструментов в виде `package testing` и команды `go test`. И с тем и с другим мы уже имели дело при тестировани функций на скорость выполнения (benchmarking), а при выполнении примеров - функций-примеров и файлов-примеров - только с командой `go test`.

Давайте посмотрим, как писАть модульные тесты с использованием Go. При этом развёртывать тесты мы будем, как и полагается, параллельно с разработкой кода.

Первый пример

Начнём с того, что напишем простенькую программку square.go:

```
package main

import "fmt"

func Square(a int) int {
    return a*a
}

func main() {
    n:= 32
    fmt.Printf("%d^2 = %d", n, Square(n))
}
```

Программка ожидаемо выведет: `32^2 = 1024` .

А теперь создадим тест для этой программы, точнее говоря, не для программы, а для функции `Square` . Сохраним её в том же каталоге в файле `square_test.go` , неожиданное название, да?

```
package main

import "testing"

func TestSquare(t *testing.T) {
    result := Square(32)
    if result != 1024 {
        t.Error("result should be 1024, got", result)
    }
}
```

В этом файле нет `func main()` , но, понятное дело, она нам и не нужна - мы зайдём в каталог, содержащий эти наши файлы, и запустим тест с помощью команды

```
go test
```

которая выведет

```
PASS
ok      Square  0.204s
```

или, лучше, дадим команду

```
go test -v
```

которая выведет

```
=== RUN   TestSquare
--- PASS: TestSquare (0.00s)
PASS
ok      Square  0.204s
```

С синтаксической точки зрения здесь важно то, что название тест-функции начинается со слова `Test` , за которым идёт большая буква, имя файла заканчивается на `_test` , а тест-функция имеет параметр типа `*testing.T` .

Интересно узнать, что это за параметр, что это за тип такой `testing.T` .

package testing

Полезли в документацию package testing `https://pkg.go.dev/testing` и увидим:
<цитата>
package testing обеспечивает поддержку автоматического тестирования пакетов *Go*. Он предназначен для использования совместно с командой *"go test"*, которая автоматизирует выполнение любой функции вида

```
func TestXxx(*testing.T)
```

где *Xxx* не начинается со строчной буквы. Имя функции служит для идентификации процедуры тестирования.

В рамках этих функций используйте методы `Error`, `Fail` или связанные с ними методы для сигнализации об ошибке.

Чтобы написать новый набор тестов, создайте файл, имя которого заканчивается *_test.go*, содержащий функции *TestXxx*, как описано здесь. Поместите файл в тот же пакет, что и тестируемый. Файл будет исключен из обычных сборок пакетов, но будет включен при выполнении команды `go test` .
<конец цитаты>

тип testing.T

Тест-функция должна иметь параметр типа `*testing.T` . Не будем здесь приводить все методы этого типа - проще и понятнее будет прочитать их в мануале по адресу `https://pkg.go.dev/testing#T` . Но пробежимся быстренько по некоторым из них.

В общем-то довольно естественно, что тест-функция не возвращает значений, ведь если тест завершается, как ожидалось, то тест проходит, и нечего тут возвращать. Но если тест не проходит, то как раз и используется параметр типа `*testing.T` .

Тест-функция заканчивается, когда она заканчивается естественным образом, либо если выполняется (вызывается) любой из методов **`FailNow, Fatal, Fatalf, SkipNow, Skip или Skipf`** . Все эти методы, также, как и метод **`Parallel`** (о нём ещё поговорим), могут вызываться только из горутины, вызывающей тест-функции. В то же время методы **`Log, Logf, Error, Errorf`** , которые также позволяют как-то сообщать о результатах тестирования, могут вызываться одновременно из нескольких горутин.

Также естественно, что раз уж тест-функция сообщает о себе только через методы, а не возвращает значения, то в тест-функциях используются проверки типа

```
if err != nil {
    t.Errorf("Unexpected error for input %v: %v", tt.input, err)
}
```

Да, писанины при таком подходе побольше, зато все возможныфе неприятности видны явно.

Про метод `Run`, позволяющий выполнять не все, а только некоторые тест-функции (управляемый параметрами команды `go test`), мы ещё поговорим. Остальные методы довольно понятны в применении.
Поиграться самостоятельно с методами типа **testing.T** - это просто обязательное **задание**.

Продолжаем построение и рассмотрение примеров

Собираем тестируемые функции и тест-функции в один пакет

файл hello.go (с функциями)

```
package hello

func hello(subj string) string {
    return "Hello, " + subj + "!"
}

func helloWorld() string {
    return "Hello, World!"
}
```

файл hello_test.go (с тест-функциями)

```
package hello

import "testing"

func TestHelloWorld(t *testing.T) {
```

```
    expected := "Hello, World!"
    actual := helloWorld()
    if actual != expected {
        t.Error("Test failed")
    }
}

func TestHello(t *testing.T) {
    expected := "Hello, Go!"
    actual := hello("Go")
    if actual != expected {
        t.Error("Test failed")
    }
}
```

Оба файла помещаем в свой каталог и, находясь в этом каталоге, создаём mod-файл

```
go mod init hello
go mod tidy
```

и подаём команду

```
go test
```

На выходе получаем

```
PASS
ok      hello   0.199s
```

Более предпочтительная команда (а в дальнейшем будем использовать только её)

```
go test -v
```

выводит

```
=== RUN   TestHelloWorld
--- PASS: TestHelloWorld (0.00s)
=== RUN   TestHello
--- PASS: TestHello (0.00s)
PASS
ok      hello   0.211s
```

Что мы увидели? Мы увидели, что

1. Тестировать можно пакеты целиком. Сразу заметим, что можно и по частям, но об этом ниже.
2. Тестировать можно как беспараметрические функции, так и функции с параметрами. К слову, в том числе и с variadic-параметрами.

Микроупражнение. Посмотреть, как команда `go test -v` реагирует на всякие "неправильности":

- Погонять тесты, "испортив" ожидаемый выход - увидеть, как тест-функции реагируют на "неправильный" результат тестируемой функции.
- Испортить текст тестируемой функции на синтаксически некорректный и убедиться, что команда `go test` в самом деле каждый раз перекомпилирует исходный текст.
- и т.д.

Тестирование пакета

Так что, названия тест-функций не привязаны к объектам тестирования? Да. Так значит мы можем в одну тест-функцию запихать тестирование многих функций? Тоже да. Сразу скажем, что это не есть хорошо, но давайте посмотрим на такой вариант - на нём покажем потребность в журналировании и приведём пример журналирования тестирования.

Возьмём наш пример с квадратом числа. Первое - изменим название пакета на `package square`, второе - уберём `func main()`, она нам вовсе ни к чему, а третье - добавим `func PerfectSquare (n int) (ok bool, sqrt int)`, которая проверяет, является ли параметр `n` точным квадратом, и, если да, то возвращает ещё и квадратный корень этого числа. Более того, напишем три варината этой функции. Отличаются они реализацией, все три реализации отличаются по эффективности (это ещё споёт при тестировании производительности, оно же бенчмаркинг, но о нём - в следующий раз, в этот раз времени точно не хватит).

package square - файл square.go:

```
package square
import (
    "math"
    "sort"
)

// Returns the square of x
```

```
func Square(x int) int {
    return x*x
}

// If n is perfect square, then PerfectSquare1 returns
// true and value of square root of n, else PerfectSquare1
// returns false and some undefined integer
func PerfectSquare1 (n int) (ok bool, sqrt int) {
    sqrt = int(math.Round(math.Sqrt(float64(n))))
    ok = Square(sqrt) == n
    return
}

// If n is perfect square, then PerfectSquare2 returns
// true and value of square root of n, else PerfectSquare2
// returns false and some undefined integer
func PerfectSquare2 (n int) (ok bool, sqrt int) {
    sum, delta := 0, 1
    for sum < n {
        // k^2 - (k-1)^2 = 2*k - 1
        sum += delta
        delta += 2
    }
    ok, sqrt = sum == n, delta/2
    return
}

// If n is perfect square, then PerfectSquare3 returns
// true and value of square root of n, else PerfectSquare3
// returns false and some undefined integer
func PerfectSquare3 (n int) (ok bool, sqrt int) {
    c := func(i int) bool {
        return i*i - n >= 0
    }
    sqrt = sort.Search(n+1, c)
    ok = Square(sqrt) == n
    return
}
```

и сразу же добавим тестирующий файл square_1_test.go:

```go
package square

import (
    "testing"
    "log"
    "os"
)

func TestPackageSquare(t *testing.T) {
    f, _ := os.Create("log.txt")
    log.SetOutput(f)
    log.SetFlags(log.Lshortfile | log.Lmsgprefix)
    defer f.Close()

    log.SetPrefix("TestPackageSquare: ")
    log.Println("Start.")

    result := Square(32)
    if result != 1024 {
        t.Error("result should be 1024, got", result)
    }

    log.SetPrefix("PerfectSquare1: ")
    ok, result := PerfectSquare1(1025) // !!!
    if !ok || result != 32 {
        log.Printf("result should be <true 32>, got <%t %d>", ok, result)
        t.Error("result should be <true 32>, got <", ok, result, ">")
    }
    if ok, _ = PerfectSquare1(1000); ok {
        t.Error("result should be false, got true")
    }

    ok, result = PerfectSquare2(1024)
    if !ok || result != 32 {
        t.Error("result should be <true 32>, got <", ok, result, ">")
    }
    if ok, _ = PerfectSquare2(1000); ok {
        t.Error("result should be false, got true")
    }
```

```
    ok, result = PerfectSquare3(1024)
    if !ok || result != 32 {
        t.Error("result should be <true 32>, got <", ok, result, ">")
    }
    if ok, _ = PerfectSquare3(1000); ok {
        t.Error("result should be false, got true")
    }

    log.SetPrefix("TestPackageSquare: ")
    log.Println("Finish.")
}
```

Здесь мы видим - тестирование всех функций собрано в одной тест-функции, причём в одном месте (оно отмечено комментарием // !!!) умышленно допустим погрешность. Давайте запустим тест. Команда

```
go test -v
```

выводит

```
=== RUN   TestPackageSquare
    square_1_test.go:27: result should be <true 32>, got < false 32 >
--- FAIL: TestPackageSquare (0.25s)
FAIL
exit status 1
FAIL    square  0.480s
```

И что? Да, тест-функция грохнулась, и много нам это даёт? Даёт мало. Что делать? А делать надо:

1. тестировать каждую единицу (в данном случае это функции, но могут быть и другие единицы, методы, например) отдельно - в отдельной тест-функции
2. журналировать тестирование, в том числе и при создании своей тест-функции для каждой тестируемой единицы (модуля, юнита - модульное же тестирование), причём журналировать следует как сбои при тестировании, так и успешно пройденные тесты

В вышеприведённом (ужасном) примере некоторое журналирование уже сделано, правда журналируется совсем немного, так, для примера. Пример ужасный - давайте исправим его. Во-первых, построим для каждой тестируемой функции свою тест-функцию:
тестирующий файл square_2_test.go:

```go
package square

import "testing"

func TestSquare(t *testing.T) {
    result := Square(32)
    if result != 1024 {
        t.Error("result should be 1024, got", result)
    }
}

func TestPerfectSquare1(t *testing.T) {
    ok, result := PerfectSquare1(1024)
    if !ok || result != 32 {
        t.Error("result should be <true 32>, got <", ok, result, ">")
    }
}

func TestPerfectSquare1_no(t *testing.T) {
    if ok, _ := PerfectSquare1(1000); ok {
        t.Error("result should be false, got true")
    }
}

func TestPerfectSquare2(t *testing.T) {
    ok, result := PerfectSquare2(1024)
    if !ok || result != 32 {
        t.Error("result should be <true 32>, got <", ok, result, ">")
    }
}

func TestPerfectSquare2_no(t *testing.T) {
    if ok, _ := PerfectSquare2(1000); ok {
        t.Error("result should be false, got true")
    }
}

func TestPerfectSquare3(t *testing.T) {
    ok, result := PerfectSquare3(1024)
    if !ok || result != 32 {
```

```
        t.Error("result should be <true 32>, got <", ok, result, ">")
    }
}

func TestPerfectSquare3_no(t *testing.T) {
    if ok, _ := PerfectSquare3(1000); ok {
        t.Error("result should be false, got true")
    }
}
```

А, во-вторых, отжурналируем этот пример. Но оставим это на **задание-упражнение**: тщательно отжурналировать пример тестирования `square_2_test.go`; при этом постараться воткнуть туда побольше всяких возможностей журналирования (`package log`) - некоторые возможности применены в примере `square_2_test.go`, но желательно поэкспериментировать и с другими, не задействованными в этом примере. Да, и ещё, конечно, поиграться с методами `testing.T.Log` и `testing.T.Logf`

И ещё один важный вопрос сбросим на **задание для самостоятельной работы**: в (ужасном) примере `square_1_test.go` мы исполняли журналирование тестирования с помощью `package log`. Тип `testing.T` имеет свои возможности журналирования - методы `t.Log` и `t.Logf`. Принципиально в них то, что они журналируют что-то только в случае сбоя при тестировании и записывают журнал в `stderr`, так что это менее универсально, чем использование `package log`. Но это средство весьма важно, так что с ним надо поиграться.

Запуск субтестов в Go

А вот и метод

```
    func(t *testing.T) Run(name string, f func(t *T)) bool
```

подоспел.
`package testing` предоставляет возможность запускать не все тесты сразу, а только некоторое подмножество тестов.

Вызвать тест-функцию по имени.

Например, переходим в каталог `square` и подаём команду:

```
go test -run TestSquare
```

запустится только функция `TestSquare`.

Передача в тестирующую программу аргументов, которые могут использоваться тест-функциями.

Заменим тестовые функции в `square_2_test.go` на такие функции:
файл square_3_test.go:

```
package square

import "testing"

func TSquare(t *testing.T) {
    result := Square(32)
    if result != 1024 {
        t.Error("result should be 1024, got", result)
    }
}

func TPerfectSquare1(t *testing.T) {
    ok, result := PerfectSquare1(1024)
    if !ok || result != 32 {
        t.Error("result should be <true 32>, got <", ok, result, ">")
    }
}

func TPerfectSquare1_no(t *testing.T) {
    if ok, _ := PerfectSquare1(1000); ok {
        t.Error("result should be false, got true")
    }
}

func TPerfectSquare2(t *testing.T) {
    ok, result := PerfectSquare2(1024)
    if !ok || result != 32 {
        t.Error("result should be <true 32>, got <", ok, result, ">")
    }
}

func TPerfectSquare2_no(t *testing.T) {
    if ok, _ := PerfectSquare2(1000); ok {
```

```
        t.Error("result should be false, got true")
    }
}

func TPerfectSquare3(t *testing.T) {
    ok, result := PerfectSquare3(1024)
    if !ok || result != 32 {
        t.Error("result should be <true 32>, got <", ok, result, ">")
    }
}

func TPerfectSquare3_no(t *testing.T) {
    if ok, _ := PerfectSquare3(1000); ok {
        t.Error("result should be false, got true")
    }
}

func TestSome(t *testing.T){
    t.Run("lala", TPerfectSquare1)
}

func TestAnotherSome(t *testing.T){
    t.Run("tratata", TPerfectSquare2)
}

func TestOther(t *testing.T){
    t.Run("tralala", TSquare)
}
```

Мы убрали все прежние тест-функции, переименовав их из `Test*` в `T*`, чтобы они больше не вызвывались командой `go test`, и добавили новые тест-функции: `TestSome`, `TestAnotherSome` и `TestOther`. Все они вызывают метод `t.Run`. Это метод принимает два аргумента: первый - некоторая строка - соответствует параметру, переданному командой `go test`, а второй - это признак имени вызываемых функций, которые и будут исполнять собственно тестирование. И теперь мы можем выборочно запускать набор тестов различными способами:

- `go test -run Some`
  Запускает каждую тест-функцию, имя которой содержит `Some`. В этом случае первый аргумент `t.Run` игнорируется, поскольку мы не передали никаких параметров, которые можно было бы использовать для сопоставления с ним.

Например, команда

```
go test -v -run Some
```

выводит

```
=== RUN   TestSome
=== RUN   TestSome/lala
--- PASS: TestSome (0.00s)
  --- PASS: TestSome/lala (0.00s)
=== RUN   TestAnotherSome
=== RUN   TestAnotherSome/tratata
--- PASS: TestAnotherSome (0.00s)
  --- PASS: TestAnotherSome/tratata (0.00s)
PASS
ok      square  0.224s
```

- `go test -run Some/lala`
  То же самое, но вызов `t.Run` будет выполняться только тогда, когда первый параметр содержит строку `lala`
  Например, команда

  ```
  go test -v -run Some/tra
  ```

  выводит

  ```
  === RUN   TestSome
  --- PASS: TestSome (0.00s)
  === RUN   TestAnotherSome
  === RUN   TestAnotherSome/tratata
  --- PASS: TestAnotherSome (0.00s)
    --- PASS: TestAnotherSome/tratata (0.00s)
  PASS
  ok      square  0.222s
  ```

Табличное тестирование

Все наши примеры тест-функция исполняли каждая ровно один тест. Как-то это неприятно. А что же делать? Так естественно же - выполнять несколько тест-примеров в одной тест-функции. При этом все примеры и ожидаемые результаты собираем в таблицу - слайс структур, содержащих входные данные и ожидаемые результаты работы тестируемой функции. Рассмотрим для примера одну тест-

функцию из файла `strings_test.go`, входящего в `package strings`. Фрагмент оформлен в виде `package index`, чтобы тест-функцию можно было запустить, и расположен в `./samples/indexrune/indexrune_test.go`:

```
package indexrune

import     (
    "testing"
    "strings"
    "unicode/utf8"
)

func TestIndexRune(t *testing.T) {
    tests := []struct {
        in   string
        rune rune
        want int
    }{
        {"", 'a', -1},
        {"", '☺', -1},
        {"foo", '☹', -1},
        {"foo", 'o', 1},
        {"foo☺bar", '☺', 3},
        {"foo☺☻☹bar", '☹', 9},
        {"a A x", 'A', 2},
        {"some_text=some_value", '=', 9},
        {"☺a", 'a', 3},
        {"a☻☺b", '☺', 4},

        // RuneError should match any invalid UTF-8 byte sequence.
        {"�", '�', 0},
        {"\xff", '�', 0},
        {"☻x�", '�', len("☻x")},
        {"☻x\xe2\x98", '�', len("☻x")},
        {"☻x\xe2\x98�", '�', len("☻x")},
        {"☻x\xe2\x98x", '�', len("☻x")},

        // Invalid rune values should never match.
        {"a☺b☻c☹d\xe2\x98�\xff�\xed\xa0\x80", -1, -1},
        {"a☺b☻c☹d\xe2\x98�\xff�\xed\xa0\x80", 0xD800, -1}, // Surrogate pair
```

```
        {"a☺b☹c☹d\xe2\x98\xff\xed\xa0\x80", utf8.MaxRune + 1, -1},
    }
    for _, tt := range tests {
        if got := strings.IndexRune(tt.in, tt.rune); got != tt.want {
            t.Errorf("IndexRune(%q, %d) = %v; want %v", tt.in, tt.rune, got, tt.wan
t)
        }
    }
}
```

Всё отлично видно: каждая запись таблицы (слайса `tests`) представляет собой полный тестовый пример с входными данными и ожидаемыми результатами. Иногда в записи таблицы включают и какую-либо дополнительную информацию, например, название теста, чтобы сделать вывод легко читаемым.

Сканируя таблицу тестовых примеров, фактический тест просто перебирает все записи таблицы и для каждой записи выполняет необходимые тесты. Тестовый код пишется один раз и пробегает по всем записям таблицы, поэтому имеет смысл очень тщательно прорабатывать текст тест-функции в части сообщений об ошибках.

Табличное тестирование на основе таблиц - это не инструмент, пакет или что-то еще, это просто способ, возможность писать более внятные тесты. Это как раз что-то из области рефакторинга тестирования.

Самое приятное в табличных тестах заключается в том, что после написания тест-функции мы можем, чаще всего, игнорировать метод тестирования и просто добавлять варианты тестов в таблицу. Кроме того, входные и ожидаемые выходные данные легко видны и связаны вместе в тестовом файле. Недостатком является, что, как уже сказано, необходимо тщательно обрабатывать сообщения об ошибках, чтобы точно определить, какой именно тестовый случай не удался. Также немного сложнее создать контрольные точки для конкретных тестовых случаев.

В приведённом примере табличный тест содержал только допустимые входы. Но хорошее тестирование предполагает и проверку реакции тестируемой функции на некорректные входные данные. Следующий пример как раз содержит и такие тесты.

Пример: перевод из римской системы счисления в десятичную и обратно; табличное тестирование.

Пакет для работы с числами в римской системе счисления `samples/roman/roman.go`

```go
package roman

import (
    "errors"
    "strings"
)

var ErrInvalidFormat = errors.New("Invalid Format")

// Converts Roman number into Decimal
func RomanToDec(roman string) (result int, err error) {
    row:= []rune(roman + ".")
    for i, r:= range row {
        switch r {
            case 'M': result += 1000
            case 'D': result += 500
            case 'C': if row[i+1] == 'M' || row[i+1] == 'D' { result -= 100 } else
{ result += 100 }
            case 'L': result += 50
            case 'X': if row[i+1] == 'C' || row[i+1] == 'L' { result -= 10 } else {
 result += 10 }
            case 'V': result += 5
            case 'I': if row[i+1] == 'X' || row[i+1] == 'V' { result -= 1 } else {
result += 1 }
            default : if r !='.' || i != len(row) - 1  { return 0, ErrInvalidFormat
 }
        }
    }
    //Проверяем результат переводом его назад в римскую систему
    if s, _ := DecToRoman(result); s != roman { return 0, ErrInvalidFormat }
    return result, nil
}

var numeral =[]int{1000, 500, 100, 50, 10, 5, 1}

// Converts Decimal into Roman
func DecToRoman (n int) (roman string, err error) {
    if n <= 0 || n >= 4000 {
        return "", errors.New("Out of range")
    }
```

```
    var letter string
    for _, v:= range numeral {
        switch v {
            case 1000: letter = "M"
            case  500: letter = "D"
            case  100: letter = "C"
            case   50: letter = "L"
            case   10: letter = "X"
            case    5: letter = "V"
            case    1: letter = "I"
        }
        for n >= v {
            n -= v
            roman += letter
        }
    }
    roman = strings.Replace(roman, "DCCCC", "CM", 1)
    roman = strings.Replace(roman, "CCCC", "CD", 1)
    roman = strings.Replace(roman, "LXXXX", "XC",  1)
    roman = strings.Replace(roman, "XXXX", "XL", 1)
    roman = strings.Replace(roman, "VIIII", "IX", 1)
    roman = strings.Replace(roman, "IIII", "IV", 1)
    return roman, nil
}
```

Табличные тесты для функции перевода римской записи числа в десятичную (`samples/roman/roman_test.go`)

```
package roman

import "testing"

var validTests = []struct {
    input    string
    expected int
}{
    {"", 0},
    {"I", 1},
    {"II", 2},
    {"III", 3},
```

```
    {"IV", 4},
    {"V", 5},
    {"VI", 6},
    {"IX", 9},
    {"X", 10},
    {"XI", 11},
    {"XIV", 14},
    {"XX", 20},
    {"XL", 40},
    {"XLIX", 49},
    {"L", 50},
    {"LIV", 54},
    {"LVI", 56},
    {"LIX", 59},
    {"LXXXVIII", 88},
    {"C", 100},
    {"CCC", 300},
    {"D", 500},
    {"M", 1000},
    {"MDCCLXXVI", 1776},
    {"MMXXII", 2022},
}

var invalidTests = []string{
    "IIX",
    "IIII",
    "VV",
    "VX",
    "XXXX",
    "LL",
    "LC",
    "CCCC",
    "DD",
    "DM",
    "Z",
    "ZI",
    "IZ",
    "123",
    "@",
    "MMMM",
```

```
}

func TestValid(t *testing.T) {
    for _, tt := range validTests {
        res, err := RomanToDec(tt.input)
        if err != nil {
            t.Errorf("Unexpected error for input %v: %v", tt.input, err)
        }
        if res != tt.expected {
            t.Errorf("Unexpected value for input %v: %v", tt.input, res)
        }
    }
}

func TestInvalid(t *testing.T) {
    for _, tt := range invalidTests {
        res, err := RomanToDec(tt)
        if err == nil {
            t.Errorf("Expected error for input %v but received %v", tt, res)
        }
        if err != ErrInvalidFormat {
            t.Errorf("Unexpected error for input %v: %v", tt, err)
        }
    }
}
```

Кажется, что обсуждать тут почти нечего. Ну разве только то, что для одной и той же функции `RomanToDec` написаны две тест-функции - одна для корректных входных данных, другая - для некорректных. В принципе, можно их обе и объединить, разве что тогда диагностика ошибки станет посложнее. Но так, как написано, выглядит вполне осмысленно - разнообразных вариантов некорректного входа для функции `RomanToDec` великое множество. Другое дело обратная функция `DecToRoman`, в ней все возможные некорректности - это подать число вне диапазона [1; 3999], так что писАть две тест-функции для этой функции вряд ли имеет смысл. Ну уж точно не две таблицы с тестами. Написать эту тест-функцию и таблицу тестов к ней - это задание. На самом деле, в этом сюжете видны сразу несколько заданий, так что давайте сформулируем их:

Задания в связи с `package roman` для работы с числами в римской системе счисления

- Написать тест-функцию и таблицу с тестами для тестирования функции

`DecToRoman`

- Функция `RomanToDec` работает не совсем честно. По крайней мере так кажется на первый взгляд - проверка корректности входных данных исполняется так: переводим входные данные в десятичную запись, полученный результат переводим обратно с помощью функции `DecToRoman` и сравниваем полученный результат с входными данными. Получается, что корректность функции `RomanToDec` базируется на корректности функции `DecToRoman`. Если мы обнаружим ошибку в `DecToRoman`, то после её исправления надо будет перетестирвать (и, возможно, исправлять) и `RomanToDec`. Как-то это не совсем хорошо выглядит, хотя вроде и нормально... Тем не менее, задание таково: написать независимую, не опирающуюся на функции `RomanToDec` и `DecToRoman`, и эффективную проверку корректности римской записи числа
- Изменить реализацию функции `RomanToDec` так, чтобы она не использовала `DecToRoman` для проверки корректности входных данных. Возможно, это будет просто обращение к функции из предыдущего пункта задания, возможно, удастся как-то эту проверку встроить в процедуру перевода римской записи в десятичную, достигая несколько большей эффективности.

И в заключение, соберём здесь вместе все

Задания для самостоятельной работы

- **Задание**. Поиграться самостоятельно с методами типа **testing.T**
- **Микроупражнение**. Посмотреть, как команда `go test -v` реагирует на всякие “неправильности” при тестировании `package hello`:
  - Погонять те же тесты, “испортив” ожидаемый выход - увидеть, как тест-функции реагируют на “неправильный” результат тестируемой функции.
  - Испортить текст тестируемой функции на синтаксически некорректный и убедиться, что команда `go test` в самом деле каждый раз перекомпилирует исходный текст.
  - и т.д.
- **задание-упражнение**: тщательно отжурналировать пример тестирования `square_2_test.go`; при этом постараться воткнуть туда побольше всяких возможностей журналирования (`package log`) - некоторые возможности применены в примере `square_2_test.go`, но желательно поэкспериментировать и с другими, не задействованными в этом примере. Да, и ещё, конечно, поиграться с методами `testing.T.Log` и `testing.T.Logf`
- **Задание для самостоятельной работы**: в (ужасном) примере `square_1_test.go` мы исполняли журналирование тестирования с помощью `package log`. Тип `testing.T` имеет свои возможности журналирования - методы `t.Log` и `t.Logf`. Принципиально в них то, что они журналируют что-то только в случае сбоя при тестировании и записывают журнал в `stderr`, так что это менее универсально, чем использование `package log`. Но это средство весьма важно, так что с ним надо поиграться.
- **Задание в связи с `package roman` для работы с числами в римской системе счисления**

- Написать тест-функцию и таблицу с тестами для тестирования функции `DecToRoman`
- Функция `RomanToDec` работает не совсем честно. По крайней мере так кажется на первый взгляд - проверка корректности входных данных исполняется так: переводим входные данные в десятичную запись, полученный результат переводим обратно с помощью функции `DecToRoman` и сравниваем полученный результат с входными данными. Получается, что корректность функции `RomanToDec` базируется на корректности функции `DecToRoman`. Если мы обнаружим ошибку в `DecToRoman`, то после её исправления надо будет перетестирвать (и, возможно, исправлять) и `RomanToDec`. Как-то это не совсем хорошо выглядит, хотя вроде и нормально... Тем не менее, задание таково: написать независимую, не опирающуюся на функции `RomanToDec` и `DecToRoman`, и эффективную проверку корректности римской записи числа
- Изменить реализацию функции `RomanToDec` так, чтобы она не использовала `DecToRoman` для проверки корректности входных данных. Возможно это будет просто обращение к функции из предыдущего пункта задания, возможно удастся как-то эту проверку встроить в процедуру перевода римской записи в десятичную, достигая несколько большей эффективности.